Formal Methods in Knowledge Engineering

Frank van Harmelen & Dieter Fensel
SWI
University of Amsterdam
Roetersstraat 15
1018MT Amsterdam
The Netherlands
frankh@swi.psy.uva.nl

September 19, 1995

Abstract

This paper presents a general discussion of the role of formal methods in Knowledge Engineering. We give an historical account of the development of the field of Knowledge Engineering towards the use of formal methods. Subsequently, we discuss the pro's and cons of formal methods. We do this by summarising the proclaimed advantages, and by arguing against some of the commonly heard objections against formal methods. We briefly summarise the current state of the art and discuss the most important directions that future research in this field should take. This paper presents a general setting for the other contributions in this issue of the Journal, which each deal with a specific issue in more detail.

1 Historical growth of Knowledge Engineering towards Formal Methods

Although the history of KBS technology and Knowledge Engineering (KE) is well documented in a number of places in the literature (e.g. [42, ch.2]), in this section we will give an account of the development of KE[1] which will show the natural growth of this field to the use of formal methods. In the development of KE methods and technology, we distinguish three main periods: the programming period, the modelling period, and the current move towards the introduction of formal methods.

The first era of KE technology stretched from the late '70s to the mid '80s. This period was characterised by the development of new programming techniques. New systems were described in terms of the representation techniques that they employed: rules, frames, Horn clauses, semantic networks, etc. KBS development environments gave support at the level of these representation techniques, and often aimed at integrating these different representations (e.g ART, KEE, Knowledge Craft, see [43] for a comparison). Such programming techniques were often developed and widely used before a proper formal

[1]At least from a European perspective. It is well possible that from an American or Japanese perspective, a different picture would emerge.

understanding of them was available (witness e.g. the certainty factor model in MYCIN [15], or multiple inheritance in LOOPS [48]). Although shells like EMYCIN [77] and others made some progress towards abstraction, they were still presented in terms of the representation language and inference engine they provided.

The move away from this first period was already heralded as early as 1980 by Alan Newell in his "knowledge level" lecture [59], but should more properly be situated as late as 1985 when Clancey published his "heuristic diagnosis" paper [17]. In this paper, Clancey analysed a number of systems at a higher level of abstraction than simply their code. In particular, he identified a specific problem solving method which underlay the behaviour of a number of systems, even though they were all coded in different ways. Clancey called this method "heuristic classification" and described it in terms of its essential inference steps and the types of knowledge manipulated by these inference steps. Most importantly, this analysis was entirely independent of the particular way this method could be programmed in any particular representation language. This type of analysis triggered the development of a number of Knowledge Engineering methodologies. The late '80s saw a number of such methodologies which were all aimed at so called "knowledge level" analysis of KBS tasks and domains. Generic Tasks [16], KADS [83, 84], Methods-to-Tasks [57] and Role-limiting methods [54] are some of the prominent examples of such methods.

These approaches all differ in the structure that they propose for analysing knowledge, the degree of task-specificity, their link with executable code, and many other properties, but all of them are based on the idea of constructing a "conceptual model" of a system which describes the required knowledge and strategies at a sufficiently high level of abstraction, independent of any particular implementation formalism.

Although these methods were highly successful and widely adopted (e.g. [73, 8, 52]), almost all of them were often criticised from both within and from outside the KBS community (e.g. from Software Engineering) for their informality and corresponding lack of precision. Even though many of these methods claimed to be based on Newell's knowledge level hypothesis, for which Newell himself had proposed logic as the ideal language for analysis, none of the most prominent methods in the late '80s incorporated much formal analysis. At best these methods offered structured and semi-formal notation without clear semantics (e.g. the inference structures of KADS), or they interfaced directly with executable inference and representation mechanisms (e.g. Generic Tasks), but none provided either a formal syntax or a mathematical semantics, let alone formal derivation rules.

This lack of formal backbone in most if not all of the leading KE methodologies was apparently felt in many places, since the third period in this historical sketch, beginning in the early '90s, brought a plethora of attempts at formalising much of the work that had been done before. The arguments that were used to motivate these formalisations will be discussed in more depth in section 3. They concerned issues like the removal of ambiguity, the possibility of formally deriving properties like soundness or completeness, and bridging the gap between an informal conceptual model and the design of an executable system. Much of this formal language development took place in Europe, and because of the prominence of the KADS conceptual model in European KE, a large number of different proposals for formalising KADS were published: MODEL-K [46], OMOS [51], MoMo [80, 31] (all from GMD in Bonn), FORKADS [81, 82] from IBM Heidelberg, $(ML)^2$ [75] from Amsterdam. Other languages were not directly based on KADS, but were based on a conceptual model closely related to KADS: KARL [4, 25] from Karlsruhe based on the

MIKE model [3], KΔBSSF [41] from the Dutch Telecom based on the VITAL model [45], and TFL [62] from Paris based on the TASK model. Yet other languages were not originally based on KADS, but it was shown how they could be used for KADS-like models: QIL [63] from Nottingham, GCLA [5] from SICS in Sweden, AIDE [47] from Compiegne and MODEL [7]. KADS was certainly not the only source of formal languages. Formal modelling languages based on other conceptual models are DESIRE [76] from the Free University of Amsterdam, MC/GETFOL [33] from Trento, and MILORD [65] from Blanes. Full references and brief comparisons of these languages will be given in section 3. All of these languages aim at capturing the prominent distinctions made in these conceptual models: distinguishing different knowledge types; describing the use of these knowledge types in inference steps; a specification of control knowledge; and aiming at task- and domain -independent descriptions. Other work applied existing formal specification languages to KBS systems or architectures. Examples include the use of Z [19, 20], CCS [79, 78] and OBJ3 [58]. The community has been actively organising itself: workshops at ECAI'92 (Vienna) and ECAI'94 (Amsterdam), as well as regular informal meetings (Bonn '92, Karlsruhe '93, Bonn '94, Amsterdam '95 and Paris '96), plus a WWW resource at `ftp://swi.psy.uva.nl/pub/keml/keml.html`. The work in this field has been extensively reviewed in [27] for KADS-based languages and in [71] for a collection of languages based on a variety of conceptual models. These reviews will be summarised in section 3.

The evolution of KE as sketched in these three periods also has direct consequences for the life-cycle employed in KBS development. In the first period, Knowledge Acquisition was seen as a direct transfer of human problem-solving expertise to a computer program. The acquired knowledge was immediately represented by a running prototype. In the second period, Knowledge Acquisition is viewed not as a transfer process but as a modelling activity. The result of Knowledge Acquisition is no longer only a running program, but a conceptual model that describes problem-solving expertise in an implementation independent manner. As a consequence, a large gap arose between the outcome of knowledge acquisition and the final implementation of a KBS. Conceptual models described in natural language are insufficient input for the implementation step, because it is mainly a question of natural language interpretation whether an implementation fulfils such a specification. The intuition of a programmer has to fill in the gaps and has to resolve ambiguity in such models. Normally it is not at all clear whether a programmer has the necessary domain and task knowledge to do this properly. The development of formal modelling techniques were a natural answer to this shortcoming by defining an intermediate level between semi-formal models and implementations. They enable a precise and detailed specification of the KBS and the required knowledge, while still abstracting in a twofold manner from the implementation. First, they abstract from implementational details (e.g. whether a set will be implemented by a list or by an array). Secondly, an implementation is normally optimised to improve its efficiency. Such efficiency aspects which are not related to the expertise but to the way it is implemented by appropriate data-structures and algorithms are of no concern during knowledge acquisition. Using a formal specification which is not used as an efficient implementation of the KBS enables to abstract from these efficiency aspects which are only related to its implementation. Using the implementation of a KBS as its specification leads either to a mix-up of these quite different aspects or leads to non-efficient implementations. Therefore, formal specification techniques save the abstraction from the implementational level as achieved by the knowledge-level hypothesis of Newel but add substance and formal preciseness to it.

It seems fair to say that the knowledge level hypothesis of Newel was a necessary step as it enabled to get rid of implementational details when discussing models of expertise. Otherwise, it caused significant irritation in its original formulation by Newel. Debates arose about what could or should by specified at the knowledge level, and about appropriate formalisms (if any). This irritation is not at all a surprise as it could not be clear from the beginning what the appropriate notion for specifying KBS's would be. Meanwhile, a number of approaches exist which characterise what should be specified at the knowledge level and what are the appropriate modelling primitives to express such models. As a consequence, work on formal specification techniques could be done which defines formal semantics for these new modelling primitives. In that way the semiformal specifications at the knowledge level can be viewed as a necessary *intermediate* step. The knowledge-level hypothesis enabled the research community to escape from the limitations of the available implementation formalism and it created the *possibility* and the *necessity* for new types of formal specification techniques. It enforced the development of formal specification techniques which reintegrate the precision and unambiguity of implementation formalisms into knowledge level specifications without losing their conceptual structure and without getting confused by implementational aspects.

The purpose of this paper is to review and discuss the motivations for this body of recent work (section 2), to briefly survey the state of the art (section 3), and to identify future research directions (section 4). Another paper in this issue deals with the relation of the formal methods in KE with their counterparts in Software Engineering and Information Systems. The final paper in this issue discusses the potential of these formal approaches for validation and verification of KBS.

2 Pro's and cons of formal methods in Knowledge Engineering

In Software Engineering, the usefulness of formal methods has been a hotly debated topic during many decades. During the recent growth of Knowledge Engineering towards the use of formal methods, many of the same debates resurfaced in the Knowledge Engineering community (e.g. [23]). In this section, we will first very briefly repeat the claimed advantages for formal methods. We will take the advantages that are claimed for formal methods in Software Engineering (see e.g. [85]), and reinterpret these in the context of Knowledge Engineering. In the second and major part of the this section, we will follow [35] and [11] by listing some of the most prominent myths about the use of formal methods in Knowledge Engineering (KE) in particular, but also in Software Engineering (SE) in general. We try to dispel most if not all of these myths.

2.1 Advantages of formal methods in Knowledge Engineering

The advantages of the use of formal methods can be distinguished by the phase of the system development to which they contribute:

Modelling: The conceptual models that are constructed in a modern Knowledge Engineering process are typically defined using a combination of natural language and graphical elements. As a result, these models are defined only informally or at best semi-formally. As is well known, such documents in natural language have an ambigous and imprecise semantics. There are as many meanings for a specification as there are readers, and it is a question of text interpretation as to whether a specification is sufficient for a system. The ambiguity of such conceptual models in a Knowledge Engineering context was aptly illustrated by Aben's analysis of the use of a single inference step from the KADS framework in a number of papers in [8]. Aben [1, p.36] states that out of nine papers that use the `abstract` inference step, only one author uses it in correspondence with the original definition in [14, p.37]. Most of the other papers use different versions of the inference step, even including versions with different numbers of input arguments. Models which are expressed in a formal specification language have a precise semantics grounded in mathematical representations and can disambiguate the informal representations. In principle, direct implementations of the informal models can also serve this purpose. However this would imply either committing the conceptual model to particular machine-oriented details (in order to obtain an efficient implementation) or to ignore such details with an efficient and badly designed implementation as a result (see also myth 1 below).

Design: Although formal specifications can often themselves not be directly efficiently executed, they can form a bridge between the informal high-level conceptual models and detailed machine-oriented design. This is because formal specifications do provide additional detail to the informal models, but still do so at a high conceptual level. An approach to the use of formal representations as a bridge between conceptual models and detailed design for KBS can be found in [50].

Furthermore, formal specification languages open the possibility for gradual and stepwise refinement of a specification towards an efficienct implementation, where each of the refinement steps is provably correct. This then guarantees the correctness of the final implementation with respect to the initial specification. Advances in this area have been made particularly with the VDM and CIP projects [10, 9].

Evaluation and maintenance: Although not much exploited in practice yet (neither in Software Engineering nor in Knowledge Engineering) the use of a system description which is at the same time precise and at a high level of abstraction would be a great asset during the maintenance of a system, provided the specification of the system is kept up to date with the evolving implemenation.

Besides these advantages related to the various phases of system development, the following points apply to each of the phases mentioned above:

Validation and verification: Because formal specifications are mathematical objects, they can be subjected to formal manipulation and proofs. This can (at least in principle) lead to proofs of desired properties such as completeness and soundness of a specification with respect to the requirements. Informally, completeness states that the entire required I/O relation can indeed be derived by the specification, while soundness states the converse: the specified I/O does not exceed the required relation. When subjected to formal analysis, these concepts break up into more refined versions (e.g. [72]). Besides these two properties, other properties are candidates for formal verification and validation, such as termination, consistency, irredundancy, etc. See for a discussion on this issue also the accompanying paper by Meseguer and Preece in this issue of the Journal [55].

Besides formal proofs, validation and verification may also take place through testing if the formal specification is executable. KBS specification languages like KARL, TFL, DESIRE and others have both a solid formal foundation and are executable (even if not efficiently). Particularly in the absence of proof techniques that scale up to specifications of realistic size, this use of "specifications as prototypes" is the only currently available practical path today to the use of formal specifications in validation and verification.

Reuse of components: Many of the Knowledge Engineering methodologies mentioned in section 1 place great emphasis on the reuse of various modelling components, varying from the reuse of standardised primitive inference steps (e.g. [1] which formalises a set of standard steps proposed in [14]), to the reuse of complex problem solving methods through libraries of predefined elements (e.g. [13]). As discussed above, [1] showed the problems in reusing informally defined primitive inference steps, and experiences in the Sisyphus II project [64] show that the situation is similar in regard to problem solving methods which are only defined informally. Only names are reused, but they refer to inferences or problem solving methods with totally different or even contradictory meanings. Development effort and costs could be noticeably reduced by the actual reuse of predefined elements with a standardised meaning which comes from their formal definition.

It is not necessary that the entire reuse process is formalised: [24] is an example of an informal analysis of a family of knowledge-based systems, where the analysis has benefited from the availability of formally defined components from which these systems were constructed (see also the discussion under myth 5 below).

2.2 False objections to formal methods in Knowledge Engineering

Myth 1: The soft knowledge of experts cannot be adequately formalised

If this myth were true, it would not only be the end of formal methods in KE, but of KE as an engineering discipline overall. After all, if it were true that soft expert-knowledge cannot be formalised, it would also apply, and to a much greater extent, to the final implementation of a KBS, because an executable computer program has for fundamental reasons a more limited expressive power than, e.g., first order predicate logic. In addition, the natural language description of the expertise itself already represents a significant reduction and we could ask with the same justification whether human expertise which is based primarily on skill can be described adequately using natural language.

[85] argues persuasively against this myth in the following way: "Programs, however, are formal objects, susceptible to formal methods ... Thus, programmers cannot escape from formal methods. The question is whether they work with informal requirements and formal programs, or whether they use additional formalism to assist them during requirements specification." [85]

Myth 2: Formal languages are difficult to learn

This is the usual argument of the Assembler programmer against FORTRAN and fundamentally boils down to the question of what is easier to understand: low-level or high-level programming languages. Every executable, and in the strictest sense, also every formal specification language can be considered as a programming language at a very high conceptual level. The aspect of an implementation which is efficient and close to the machine is here less important than the interest in offering language constructs at a high conceptual level. Zave, who was deeply involved in the development of the operational specification language PAISley, referred to the fact that: "an executable specification language is a specialized programming language" [86]. The relationship between specification languages and programming languages has changed continually. Or, formulated differently: the current specification languages are the programming languages of the future, as the higher conceptual level allows programming which is more efficient because it is more understandable.

The above argument holds for SE in general, but KE itself is in an even stronger position to dispel this myth. The formal languages developed for KE use the model of expertise as a conceptual model. They allow a graphic representation of most modelling primitives and the formal specification can tie onto already existing semi-formal models and refine them. The graphic modelling primitives of the KADS model are supplemented in this, for example, by Petri networks or extended entity relationship diagrams (EER).

Myth 3: Formal specifications are too complex and difficult to understand

Basically, the aspects of precision and clarity are discussed here. Because formal and executable specifications force a detailed formulation of the modeled knowledge, formal specifications soon become very large and confusing. Structuring, hierarchization, and modularization are well-known techniques for moderating this problem. The conceptual models used by KE languages offer clear advantages, particularly in this point, as they differentiate clearly between different types or levels of knowledge and subdivide the entire specification into small elements with clearly defined purposes (e.g. an elementary inference action, a domain view of an inference action, a task, a knowledge role, etc.). This again is a clear advantage of the KE languages discussed here over the general-purpose specification languages of SE with their weaker conceptual model. We must always emphasize that a formal specification cannot replace an informal specification, but is intended to refine or supplement it. It is almost always easier to understand an idea by reading a natural language text than to attempt to extract it laboriously from lots of formulae. On the other hand, the attempt at finding a precise and complete natural language definition for a complex problem generally leads to illegible and incomprehensible multi-clause sentences which always contain ambiguities, redundancies, breaks, and contradictions.

A further rebuttal of this point resembles the argument against Myth 1: if formal specifications are too complex and difficult to understand, then this must surely apply with equal if not more force to the lower level implementation languages used in everyday practice, since the programs expressed in these languages are often an order of magnitude larger than the corresponding specification in a high level formal language.

Myth 4: Formal methods are all about proving programs correct

The point is made in [35] that the value of formal specifications is often not in the final product (the actual specification text), but rather in the process of creating that product. This process, with its enforced precision and detail, reveals many insights into the nature of the system to be produced, and it are these insights, rather than the actual final document, which justifies the formal specification effort. In industrial practice, it turns out that the benefit obtained during the process of formal specification are already so crucial that large specification projects report the successful use of formal methods without ever doing a single line of proof (again, see [35]). It is important not to see specifications only as a useful product, but to see the writing of specifications as an important process. In this process, statements are made more precise, ambiguities and contradictions are recognized and eliminated. The errors and problems which are recognized can be used to improve the informal specifications. Thus occasionally a formal specification is not necessary as a final document, but only the process of formalization and the improvement of the informal specification which it enables. Wing says on this subject: "The greatest benefit in applying a formal method often comes from the process of formalizing it rather than from the end result." [85]

Myth 5: Formal methods are too expensive and time consuming to use

In fact the formalization of a specification does generate additional costs. On the other hand, this objection is probably the most superficial. Every standard textbook of SE is full of admonitions that the later a specification or programming mistake is discovered in the software development process, the more expensive it will be to eliminate it. An informal specification with its basic incompleteness forces the programmer literally to bridge gaps using his imagination. And he or she does not always meet the expectations of the later users or the lacking expert knowledge.

The predominant share of the effort in the construction of a formal specification is thus not due to the use of a formal specification but rather to the goal of making an informal specification more precise and eliminating ambiguities and contradictions. If the system is to be implemented, then this effort must be expended anyway. But this will then no longer be carried out in an appropriate phase and by the appropriate persons. "The fact is that writing a formal specification decreases the cost of development." [35]

Besides these general points, again KE is in a particularly strong position to dispel this myth. As shown in [74], the close correspondence between the informal conceptual model and the formal specification language in KE can be exploited to give a much stronger support (both automated and non -automated) to the transition from informal to formal model, and such support is likely to significantly reduce the cost associated with formal specification.

Finally, the KE community has traditionally placed great emphasis on the possibility to *re-use* library elements or fragments of existing models in the construction of new models. Such re-use is only properly possibly if the fragments are indeed specified precisely enough to be reusable. In the context of Knowledge Engineering, these fragments will consist of the contents or structure (ontology) of domain-knowledge, elementary inference steps [1] or entire problem-solving methods [13]. Each of these fragments must be accompanied by a precise characterisation of the assumptions it makes about the knowledge, data and computational environment required for its proper functioning. Re-use of such fragments will then (among other things) require a (preferably formal) proof that the requirements of a particular fragment are indeed fulfilled in the given circumstances. Such an increased possibility of reuse will be a great pay -back for the additional effort involved in building the final specification of the building blocks.

Myth 6: Formal methods never worked in SE, so why would they work in KE

First of all, the premise of this myth is increasingly untrue as time proceeds. Highly publicized accounts of the application of formal methods to a number of well-known systems, e.g. the Darlington Nuclear Facility in the UK [2, 60], the CICS project at IBM-UK [40] and Airbus [61], all reported in [39]. References [22] and [21] provide a useful survey of the use of formal methods in SE.

Secondly, as already stated above, the formal languages recently developed for KE purposes are tightly coupled with the underlying conceptual models developed earlier in the field. This has as advantages (i) that these languages can give strong support for the construction of a formal specification and (ii) that the structural correspondence can be used to verify that the formal specification does indeed capture the informally stated requirements.

The same argument about the close correspondence between informal and formal models can also be used to rule out the following:

Myth 7: Formal methods are unacceptable to users

In particular the graphical representations available for many formal languages in KE (see e.g. MIKE [3]) and the possibility to interpret the formal constructions in terms of the underlying conceptual model form a significant bridge to users and domain experts alike.

Myth 8: Formal methods are only useful in safety-critical systems

It is true that safety-critical applications can gain obvious benefits from formal methods. We quote the following from [11]:

"The UK Ministry of Defence (MoD) draft Interim Defence Standards 00-55 and 00-56 mandate the extensive use of formal methods. Standard 00-55 sets forth guidelines and requirements; the requirements include the use of a formal notation in the specification of safety-critical components, and an analysis of such components for consistency and completeness. All safety-critical software[2] must also be validated and verified; this includes formal proof and rigorous (but informal) correctness proofs, as well as more conventional static and dynamic analysis. Standard 00-56 deals with the classification and hazard analysis of the software and electronic components of defence equipment, and also mandates the use of formal methods".

The Atomic Energy Control Board (AECB) in Canada in conjunction with David Parnas at McMaster University has commissioned a proposed standard for software for computers in the safety systems of nuclear power stations. Ontario-Hydro has developed a number of standards and procedures within the framework set by AECB and further procedures are under development.

[2]Safety critical software is defined as software which *on its own* could cause a significant accident.

Many reported applications concern safety-critical systems, for example [39] mentions railway signaling and railway traffic tracking, Airbus cabin communication systems, besides some of the applications mentioned above. Nevertheless, non-safety-critical applications have been reported as well: [39] mentions instrumentation systems, telephone switching systems, secure operating systems and microprocessors. Famous examples of which include the T800 Transputer floating-point unit and parts of the T9000 transputer pipeline architecture (all of these reported in [11]). These real-life examples also dispel the following:

Myth 9: Formal methods are not used on real large-scale systems

Besides the extensive lists of applications mentioned above, it is important to realise here that formal methods need not necessarily be applied to entire systems. It is often possible to isolate crucial, complicated or critical components of a large system, and to limit the use of formal methods to these subsystems.

We must admit at this point however that the applications listed above concern formal methods from SE. Formal methods in KE have not yet reached the degree of maturity that they have made their way into industrial applications, and this must be seen as one of the major challenges of the field in the years to come.

3 Current state of the art

The work on formal methods in KE has been recently and extensively surveyed in [71] and [27], and we refer the interested reader to these works for detailed comparisons between the different approaches. In this section, we will instead give a broader, less detailed and also somewhat more recent overview of the current state of the field. Based on this overview, in the final section of this paper we will identify the most important and urgent points on the agenda of this community.

The work in the field until now is best summarised by the catalogue of formal languages that have been developed in recent years. Three groups of work can be distinguished here: (i) formal languages based on the KADS conceptual model, (ii) formal languages based on other conceptual models, and (iii) the use of formal languages from Software Engineering.

The languages of group (i) have been surveyed and compared in detail in [27]. That paper compares languages by looking at their operational aspects, their epistemological commitments and their formal underpinning. A very brief survey of this paper is given in table 1. Analysis in this paper shows that a major determinant of choices made in the various languages was whether a language was intended to be operational or not. Some languages (OMOS, MODEL-K, MoMo) aim at clarifying the meaning of conceptual models through operationalising such models, and do not aim at providing a formal semantics. Other languages ($(ML)^2$, QIL, KΔBSSF) aim at formalising conceptual models, but use such expressive formal constructions that this precludes effective executability. This non-executability is not only a matter of efficiency, but often also of the much more principled issue of (semi)-decidability. A third group of languages (KARL, FORKADS) aims to combine executability and formalisation. This main aim of language was found to explain

[3]This column indicates whether the language is still the subject of active research, development and use, or whether work on the language has stopped.

language	site	language constructs	executable	static semantics	dynamic semantics	active[3]
OMOS [51]	GMD, Bonn	frames, hierarchies, predicates, functions	yes	no	no	no
MODEL-K [46]	GMD, Bonn	frames, hierarchies, predicates, functions	yes	no	no	no
MoMo [80, 31]	GMD, Bonn	Petri nets, functions, arbitrary KR-languages	yes	no	no	yes
FORKADS [81, 82]	IBM, Heidelberg	sorted logic	yes	Tarskian models	no	no
KARL [4, 25]	Univ. of Karlsruhe	F-logic Horn logic Dynamic logic	yes	perfect Herbrand models	Kripke models	yes
$(ML)^2$ [75]	Univ. of Amsterdam	sorted logic meta-logic dynamic logic	no	Tarskian models	Kripke models	yes
QIL [63]	Univ. of Nottingham	predicate logic temporal logic epistemic logic	no	Tarskian models	Kripke models	yes
KΔBSSF [41]	PTT Labs Netherlands	algebra, sorted logic procedures	no	Tarskian models, initial algebras	Plotkin style	no
TFL [62]	Univ. de Paris Sud	algebraic data types	no	loose algebraic semantics	loose algebraic semantics	yes
GCLA [5]	SICS, Sweden	generalised logic programs	yes	partial inductive definitions	partial inductive definitions	yes

Figure 1: A brief survey of languages that can be used to operationalise and formalise KADS models of expertise.

many of the differences found in the detailed comparisons between the languages presented in [27].

This detail in the comparison was possible because all the languages discussed in [27] were based on the same underlying conceptual model. This was not the case for the languages compared in [71]. The basis of that comparison was a specific task- and domain-model (a simple time -table scheduling task) which was modelled in a number of languages. Again, the purpose of the language (operationalisation, formalisation or a combination of these two) was found to be of prime importance. The following common properties were found to hold across the set of languages discussed in [71]:

- the composition of a complex specification out of components which are each declaratively specified.
- distinctions between static and dynamic aspects of a specification
- distinctions between generic and domain-specific parts of the specification.

A coarse comparison was done among the languages, based on three dimensions:

Expressive power for domain knowledge: The main distinction here was between languages which used a sublanguage of first-order predicate logic (such as Horn logic) and those which used full first-order logic. Other languages which were used (such as modal logic) could in principle be encoded within first-order order logic so that from a strict semantic point of view, these languages are equivalent.

Flexibility of reasoning patterns: AIDE has a completely fixed structure of reasoning pattern, $(\mathrm{ML})^2$ and KARL have a fixed overall structure but can be configured within this structure: the three-layer structure of these languages (based on KADS) is fixed, but they are configurable within each of these three layers. MC and DESIRE are fully configurable and impose no fixed configuration.

Expressiveness of control knowledge: [71] agreed with [27] that this is the main point of difference between the various languages.

A more detailed comparison was also carried out in [71], and is summarised in figure 2

The two survey papers discussed above both focussed on novel specification languages specifically designed for use in Knowledge Engineering. Another (although much smaller) body of work has concentrated on using existing languages from Software Engineering to problems in Knowledge Engineering. Examples of this approach are:

[19, 20]: specifications of a blackboard architecture (CASSANDRA) and a production-rule architecture (including meta-rules, ELEKTRA), using Z [66, 67].

[79, 78]: specifications of a large number of blackboard architectures using CCS [56].

[58]: specification of a simple scheduling task (from [71]) using OBJ3 [30].

[69]: specification of diagnostic reasoning patterns using Z.

[49]: specification of medical knowledge models using Z.

	$(ML)^2$ [75]	MC [32]	AIDE [47]	KARL [4, 25]	DESIRE [76]	OBJ3 [58]	MILORD [65]	KΔBSSF [41]
1	FOL meta-logic dyn. logic	FOL meta-logic	restricted FOL	Horn logic, dyn. logic	3-valued FOL meta-logic temp. logic	order sorted algebra	multi- valued logic	sorted logic + algebra + proc. lang.
2	yes	yes	only at domain level	yes	component- wise. temp. composition	only at domain level	locally	only at domain level
3	yes	by the user	yes	yes	yes	no	yes	yes
4	yes	yes	yes	yes	yes	yes	yes	yes
5	yes	yes	no	yes	yes	yes	yes	hard
6	no	yes	no	no	yes	no	yes	via para- meterised deduction
7	partial	no	limited	yes	partial	yes	yes	partial
8	yes	yes	partial	yes	yes	yes	yes	yes

1. Expressive power
2. Declarativeness
3. Adequacy to specify dynamic aspects of reasoning patterns
4. Possibility to specify multi-level architectures
5. Adequacy to specify non-classical reasoning
6. Possibility to specify integrated systems
7. Executability ("partial" means that the language has an executable subset)
8. Availability of formal semantics

Figure 2: Detailed comparison in [71]

4 Future research directions

As the concluding section of this survey paper, we will sketch what we see as the most important and/or challenging research problems for this community in the next few years.

Formalisms for dynamics

As we have seen above, both survey papers [71] and [27] identified formalisms for dynamics as the cause of the largest differences between the currently available specification languages. In Software Engineering and Information Systems Design we also see a wide variety of formalisms to deal with dynamic behaviour of systems (e.g. evolving algebras [34], dynamic logic [36] and variants thereof [68], Petri nets [44], temporal logics [70]). These are discussed in some detail in the accompanying paper by Fensel [26]. It is at the moment unclear what the strengths and weaknesses of the various formalisms are, and what the actual requirements of Knowledge Engineering specification languages are in this respect.

Semantics

Connected with the open question on appropriate formalisms for dynamic behaviour of KBS is the semantics provided for such formalisms. Again, the set of currently available languages differ widely in this respect, and even more options are available in the literature on Software Engineering and on Logic.

A second problem with the semantics of these languages is the following: almost all languages view a complex specification as built out of simpler components in a structured way. Good traditional semantic foundations exist for most of the individual components employed in these languages, but it is often unclear how the overall semantics should be composed out of these individual components. Foundational work is required for a clear view on such compositional semantics.

Proof calculi

Clearly related to the problem with semantics is the development of proof calculi for these languages. Notwithstanding our reply to myth 4 above, in order to reap the full benefits of these formal specification languages, we need proof calculi which allow proving general properties of specifications. In general, a proof calculus is required which enables us to prove properties of the form

$$\textit{preconditions} \wedge \textit{specification} \rightarrow \textit{post-conditions},$$

where *preconditions* and *post-conditions* are sets of first-order formulae, and *specification* is a formal specification which includes static as well as dynamic aspects of a system. That is, *specification* is a set of formulae in a logic which includes states and changes of states. Examples for this type of logic are dynamic and temporal logic. A formal reasoning calculus together with automated proof support elicits the full power of formalisation as it becomes possible to prove implicit properties of a specified system (e.g. [37, 38]). Approaches in Software Engineering like VDM already deliver good results in this area [10].

Validation and Verification

Closely related to such proofs of properties of a specification is the use of these specification languages for validation and verification. This defines a strong link to work done in program verification (see [18, 28] for a survey and [38] for an approach using theorem proving in dynamic logic), and to work done in validating and verifying KBS (cf. [6, 53]). Until now, most of the work in validating and verifying knowledge-based systems is focused on specific implementation formalisms but this could change now as these techniques can be applied to conceptual models when these models are supplemented by a formal language and semantics. Furthermore, the current techniques only apply to the static semantics of knowledge, and do not take into account the control structures under which this knowledge is to be used. The paper by Meseguer and Preece in this issue provides a discussion on the relationship between conceptual modelling, formal methods and validation and verification of KBS.

Interaction with environment

A weak aspect of many if not all of the current Knowledge Engineering specification languages is a limitation already inherent in the conceptual models on which these languages are based. This concerns the modelling of the interaction of a system with its environment. Issues such as interaction with a user, interaction with asynchronous events and interaction with other software systems in a heterogeneous environment have all remained very underdeveloped until now. The task- and cooperation model in KADS [12] is a step in this direction, but not nearly as fully developed as the corresponding conceptual model of expertise. Work on DESIRE also has taken the interaction with the environment into account and is ahead of KADS in this respect.

This is all the more urgent since each of these points is likely to be a significant factor in many application areas. Extensive work remains to be done here, both at the conceptual and at the formal level.

Exposure to applications

In our replies to myths 6 and 8, we were forced to use almost exclusively examples from applications of formal methods in Software Engineering rather than Knowledge Engineering. The reason for this is of course that the use of formal methods in Knowledge Engineering has simply not yet reached the stage of industrial application. Of all the open points of future work in this section, this point is perhaps the most important one. After the development of a variety of formal specification languages, many of which now seem to be stable, the time is ripe for proving the value of these languages in real-life applications. Areas where KBS are already used and which seem appropriate for the use of these formal techniques are the process-industry (e.g. chemical plants, energy production), health (e.g. medical decision-making [29]), transport and finance.

Acknowledgements

We are grateful to Jan Treur, Pascal van Eck and two anonymous referees for their comments on this paper.

References

[1] M. Aben. *Formal Methods in Knowledge Engineering.* PhD thesis, University of Amsterdam, Faculty of Psychology, February 1995. ISBN 90-5470-028-9.

[2] S. Anderson and G. Bruns. The formalization and analysis of a communication protocol. In *Applications of Formal Methods* [39].

[3] J. Angele, D. Fensel, D. Landes, S. Neubert, and R. Studer. Model-based and incremental knowledge engineering: The MIKE approach. In J. Cuena, editor, *Knowledge Oriented Software Design*, number A-27 in IFIP Transactions, Amsterdam, 1993. North Holland.

[4] J. Angele, D. Fensel, D. Landes, and R. Studer. The knowledge-based acquisition and representation language KARL. 1995. (submitted for publication).

[5] M. Aronsson, M. Eriksson, and P. Kreuger. GCLA and a sketch of its use for representing KADS models. In *Proceedings of the ECAI'94 Workshop on Formal Specification Methods for Knowledge-Based Systems*, Amsterdam, August 1994.

[6] M. Ayel and J.-P. Laurent (eds.). *Validation, Verification and test of knowledge-based system.* John Wiley & Sons, Chicester, 1991.

[7] M. Barbuceanu. Towards integrated knowledge modeling environments. *Knowledge Acquisition*, 5(3), 1993.

[8] C. Bauer and W. Karbach, editors. *Proceedings Second KADS User Meeting*, ZFE BT SE 21, Otto-Hahn Ring 6, D-8000 Munich 83, 17-18 February 1992. Siemens AG.

[9] F. Bauer, H. Ehler, R. Horsch, B. Moller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP, vol II: The Transformation System CIP-S*, volume 292 of *Lecture Notes on Computer Science.* Springer Verlag, Berlin, 1987.

[10] J. Bicarregui, J. Fitzgerald, P. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practioner's Guide.* Springer Verlag, 1994.

[11] J. Bowen and M. Hinchey. Seven more myths of formal methods. In M. Naftalin, T. Denvir, and M. Bertran, editors, *Proceedings of the 2nd International Symposium of Formal Methods Europe (FME'94*, volume 873 of *Lecture Notes in Computer Science*, pages 105–115, Barcelona, October 1994. Springer Verification.

[12] J. A. Breuker and P. de Greef. Modelling system-user cooperation in KADS. In A. Th. Schreiber, B. J. Wielinga, and J. A. Breuker, editors, *KADS: A Principled Approach to Knowledge-Based System Development*, pages 47–70. Academic Press, London, 1993.

[13] J. A. Breuker and W. Van de Velde, editors. *The CommonKADS Library for Expertise Modelling.* IOS Press, Amsterdam, The Netherlands, 1994.

[14] J. A. Breuker, B. J. Wielinga, M. van Someren, R. de Hoog, A. Th. Schreiber, P. de Greef, B. Bredeweg, J. Wielemaker, J. P. Billault, M. Davoodi, and S. A. Hayward. Model Driven Knowledge Acquisition: Interpretation Models. ESPRIT Project P1098 Deliverable D1 (task A1), University of Amsterdam and STL Ltd, 1987.

[15] B. G. Buchanan and E. H. Shortliffe. *Rulebased Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project.* Addison Wesley, Reading, Massachusetts, 1984.

[16] B. Chandrasekaran. Generic tasks in knowledge based reasoning: High level building blocks for expert system design. *IEEE Expert*, 1(3):23–30, 1986.

[17] W. J. Clancey. Heuristic classification. *Artificial Intelligence*, 27:289–350, 1985.

[18] T.R. Colburn, J.H. Fetzer, and T.L. Rankin (eds.). *Program Verification.* Kluwer Academic Publisher, Dordrecht, 1993.

[19] I. Craig. *The formal specification of advanced AI architectures.* Lecture Notes in Mathematics. Ellis Horwood, 1991.

[20] I. Craig. The formal specification of ELEKTRA. Technical report, Department of CS, University of Warwick, 1991.

[21] D. Craigen. Applications of formal methods: Observations and trends. In *Applications of Formal Methods* [39].

[22] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical report, U.S. Department of Commerce, Technology administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, USA, March 1993.

[23] D. Fensel. Sinn und unsinn formaler spezificationssprachen fuer wissensbasierte systeme. *Kuenstliche Intelligenz*, (4), 1994. In German.

[24] D. Fensel. A case study on assumptions and limitations of a problem solving method. In *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW'95)*, Banff, Canada, February 1995.

[25] D. Fensel. *The Knowledge-Based Acquisition and Representation Language KARL.* Kluwer Academic Pubblisher, 1995.

[26] D. Fensel. Specification languages in knowledge engineering and software engineering. *Knowledge Engineering Review*, this volume(this number), 1995.

[27] D. Fensel and F. van Harmelen. A comparison of languages which operationalise and formalise KADS models of expertise. *The Knowledge Engineering Review*, 9:105–146, 1994.

[28] J.H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31, September 1988.

[29] J. Fox. On the soundness and safety of expert systems. *Artificial Intelligence in Medicine*, 5:159–179, 1993.

[30] K. Futatsugi, J. Goguen, J. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. K. Reid, editor, *Proc. Twelfth Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[31] Friedrich Gebhardt. MoMo: language and case studies. Fabel-Report 36, GMD, Sankt Augustin, July 1995.

[32] E. Giunchiglia and P. Traverso. A multi-context architecture for formalizing complex reasoning. *International Journal of Intelligent Systems*, 10(5):501–539, May 1995.

[33] E. Giunchiglia, P. Traverso, and F. Giunchiglia. Multi-context systems as a specification framework for complex reasoning systems. In J. Treur and Th. Wetter, editors, *Formal Specification of Complex Reasoning Systems*. Ellis Horwood, 1993.

[34] Y. Gurevich. Evolving algebras. a tutorial introduction. In E.G. Rozenberg et al., editor, *Current Trends in Computer Science*. World Scientific, 1993.

[35] J. A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[36] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol. II: extensions of Classical Logic*, pages 497–604. Reidel, Dordrecht, The Netherlands, 1984.

[37] M. Heisel, W. Reif, and W. Stephan. Implementing verification strategies in the KIV-system. In R. Lusk and R. Overbeek, editors, *9th International Conference On Automated Deduction (CADE'88)*, number 310 in Lecture Notes in Computer Science, pages 131–140, Argonne, Illinois, 1988. Springer Verlag.

[38] M. Heisel, W. Reif, and W. Stephan. Tactical theorem proving in program verification. In *Proceedings of the 10th International Conference on Automated Deduction (CADE'90)*, volume 449 of *Lecture Notes in Computer Science*. Springer Verlag, July 1990.

[39] M. Hinchey and J. Bowen (eds). *Applications of Formal Methods*. International Series in Computer Science. Prentice-Hall, 1995.

[40] J. Hoare. Formal development of CICS with B. In *Applications of Formal Methods* [39].

[41] L. in 't Veld, W. Jonker, and J.W. Spee. The specification of complex reasoning tasks in KΔBSSF. In Treur and Wetter [71], pages 233–256.

[42] P. Jackson. *Introduction to Expert Systems*. Addison Wesley, Reading, Massachusetts, second edition, 1990.

[43] J.P. Jaurent, J. Ayel, F. Thome, and D. Ziebelin. Comparative evaluation of three expert system development tools: KEE, knowledge craft, ART. *Knowledge Engineering Review*, 1(4):19–29, 1986.

[44] K. Jensen. Coloured petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986 Part I*, volume 254 of *Lecture Notes of Computer Science*, pages 248–299. Springer, Berlin, Germany, 1987.

[45] W. Jonker, J.W. Spee, L. in t Veld, and M. Koopman. Formal approaches towards design in SE and their role in KBS design. In *Proceedings of the IJCAI'91 workshop on Software Engineering for Knowledge-Based Systems*, Sydney, Australia, August 1991.

[46] W. Karbach, A. Voß, R. Schukey, and U. Drouwen. Model-K: Prototyping at the knowledge level. In *Proceedings Expert Systems–91*, pages 501–512, Avignon, France, 1991.

[47] G. Kassel and C. Greboval. How AIDE succeeds in an example design task. In J. Treur and Th. Wetter, editors, *Formal Specification of Complex Reasoning Systems*. Ellis Horwood, 1993.

[48] S. E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.

[49] P. Krause, J. Fox, M. O'Neill, and A. Glowinski. Can we formally specify a medical decision support system. *IEEE Expert*, 8(3):56–61, 1993.

[50] D. Landes and R. Studer. The treatment of non-functional requirements in MIKE. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, Barcelona, 1995.

[51] Marc Linster. Using the operational modeling language OMOS to represent KADS conceptual models. *Knowledge Acquisition*, 1993.

[52] C. Löckenhoff, D. Fensel, and R. Studer, editors. *Proceedings of the Third KADS User Meeting*, ZFE BT SE 21, Otto-Hahn Ring 6, D-8000 Munich 83, 8-9 March 1993. Siemens AG.

[53] T. Lydiard. Overview of current practice and research initiatives of the verification and validation of KBS. *Knowledge Engineering Review*, 7(2):101–113, 1992.

[54] S. Marcus, editor. *Automatic knowledge acquisition for expert systems*. Kluwer, Boston, 1988.

[55] P. Meseguer and A. Preece. Verification and validation of knowledge based systems with formal specifications. *Knowledge Engineer Review*, this Volume(this number), 1995.

[56] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, London, 1989.

[57] M. A. Musen. *Automated Generation of Model-Based Knowledge-Acquisition Tools*. Pitman, London, 1989. Research Notes in Artificial Intelligence.

[58] A. Nakagawa, T. Sakakihara, and K. Futatsugi. Algebraich specification of reasoning systems. In J. Treur and Th. Wetter, editors, *Formal Specification of Complex Reasoning Systems*. Ellis Horwood, 1993.

[59] A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.

[60] D. Parnas. Using mathematical descriptions in the inspection of safety-critical software. In *Applications of Formal Methods* [39].

[61] J. Peleska, U. Hamer, and H.-M. Hoercher. The airbus a330/340 cabin communication system – a Z application. In *Applications of Formal Methods* [39].

[62] C. Pierret-Goldbreich and X. Talon. An algebraic specification of the dynamic behaviour of knowlegde-based systems. *The Knowledge Engineering Review*, 1995. Submitted.

[63] N. Shadbolt S. Aitken, H. Reichgelt. Representing kads models in QIL. Working Paper WP-006, AI Group, University of Nottingham, 1992.

[64] A. Th. Schreiber and P. Terpstra. Sisyohus-VT: A CommonKADS solution. *International Journal of Human-Computer Studies*, 1995. In press.

[65] C. Sierra and L. Godo. Specifying simple scheduling tasks in a reflective and modular architecture. In J. Treur and Th. Wetter, editors, *Formal Specification of Complex Reasoning Systems*. Ellis Horwood, 1993.

[66] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, January 1988.

[67] J. M. Spivey. *The Z Notation: A Reference Manual*. Series in Computer Science. Prentice Hall International, 2nd edition, 1992.

[68] P. Spruit, R. Wieringa, and J.-J. Meyer. Axiomatization, declarative semantics and operational semantics of passive and active updates in logic databases. *Journal of Logic and Computation*, 5(1), 1995.

[69] B. Todd and R. Stamper. The formal design and evaluation of a variety of medical diagnostic programs. Technical Monograph PRG-109, Oxford University Computing Laboratory, September 1993.

[70] J. Treur. Temporal semantics of meta-level architectures for dynamic control. In L. Fribourg and F. Turini, editors, *Logic Program Synthesis and Transformation – Meta-Programming in Logic (LOPSTR'94 – META'94)*, volume 883 of *Lecture Notes in Computer Science*, pages 353–376. Springer Verlag, 1994.

[71] J. Treur and Th. Wetter, editors. *Formal Specification of Complex Reasoning Systems*, Workshop Series. Ellis Horwood, 1993.

[72] J. Treur and M. Willems. A logical foundation for verification. In T. Cohn, editor, *Proceedings of the 11th European Conference on AI (ECAI'94)*, pages 745–749, Amsterdam, August 1994. J. Wiley.

[73] B. Ueberreiter and A. Voß, editors. *Materials KADS User Meeting, February 14/15 1991*. Siemens AG ZFE IS INF 32, Munich Perlach, Germany, 1991. In German.

[74] F. van Harmelen and M. Aben. Structure preserving specification languages for knowledge-based systems. *International Journal of Human Computer Studies*, 1995. (Formerly Journal of Man Machine Studies).

[75] F. van Harmelen and J. R. Balder. $(ML)^2$: a formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1), 1992. Special issue: 'The KADS approach to knowledge engineering', reprinted in *KADS: A Principled Approach to Knowledge-Based System Development*, 1993, Schreiber, A.Th. *et al.* (eds.).

[76] I. A. van Langevelde, A. W. Philipsen, and J. Treur. Formal specification of compositional architectures. In B. Neumann, editor, *Proceedings ECAI'92, Vienna*, pages 272–276, Chichester, 1992. Wiley. Longer version available as: Report IR-282, Mathematics and Computer Science, Free University of Amsterdam.

[77] W. van Melle. A domain independent production rule system for consultation programs. In *IJCAI–79*, pages 923–925, 1979.

[78] H. Velthuijsen. *The nature and applicability of the blackboard architecture.* PhD thesis, University of Limburgh, Maastricht, The Netherlands, 1992.

[79] H. Velthuijsen and P. Braspenning. A conceptual and formal study of the blackboard architecture. In E. Moskilde, editor, *Proceedings of the European Simulation Multiconference (ESM'91)*, pages 374–379, Copenhagen, 1991. The Society for Computer Simulation.

[80] H. Voss and A. Voss. Reuse-oriented knowledge engineering with MoMo. In *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering (SEKE'93)*, San Francisco, June 1993.

[81] T. Wetter. First-order logic foundation of the KADS conceptual model. In B. J. Wielinga, J. Boose, B. Gaines, G. Schreiber, and M. van Someren, editors, *Current trends in knowledge acquisition*, pages 356–375, Amsterdam, The Netherlands, May 1990. IOS Press.

[82] Th. Wetter and W. Schmidt. Formalization of KADS interpretation models. In L. Steels and B. Smith, editors, *AISB91, Artificial Intelligence and Simulation of Behaviour*. Springer-Verlag, 1991.

[83] B. J. Wielinga and J. A. Breuker. Models of expertise. In *Proceedings ECAI–86*, pages 306–318, 1986.

[84] B. J. Wielinga, A. Th. Schreiber, and J. A. Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5–53, 1992. Special issue 'The KADS approach to knowledge engineering'. Reprinted in: Buchanan, B. and Wilkins, D. editors (1992), *Readings in Knowledge Acquisition and Learning*, San Mateo, California, Morgan Kaufmann, pp. 92-116.

[85] J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):10–24, September 1990.

[86] P. Zave. An insiders evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 18(3):212–225, 1991.